結構化程式的開發 3

學習目標

在本章中，你將學到：

- 解決問題的基本技術。
- 經由從上而下，逐步改良的過程來發展演算法。
- 使用選擇敘述式 `if` 以及 `if-else` 來選擇要執行的動作。
- 使用 `while` 重複敘述式來重複地執行程式中的敘述式。
- 計數器控制重複結構和警示訊號控制重複結構。
- 結構化程式設計。
- 遞增、遞減及指定等運算子。

本章綱要

3.1 簡介

- 在撰寫程式解決某個問題之前，最好能夠先充分了解問題，並仔細地規劃解決這個問題的方法。
- 在接下來的兩章中，我們將介紹有關開發結構化電腦程式的一些技巧。

3.2 演算法

- 任何計算的問題均可歸納成以特定的順序來執行一系列的動作。
- 於是我們可列出解決問題的程序(procedure)如下
  - 將要執行的動作 (actions)，以及
  - 執行這些動作的順序 (order)。
- 這就稱為是演算法 (algorithm)。
- 執行這些動作的順序(order)是很重要的。

3.2 演算法

- 讓我們來看看下面這個演算法，它敘述一個中級主管一大早起床後所該做的事：讓我們來看看下面這個演算法，它敘述一個中級主管一大早起床後所該做的事：(1) 起床 (2) 脫掉睡衣 (3) 洗澡 (4) 穿好衣服 (5)吃早餐 (6)開車上班。
- 這個程序能夠讓這個中級主管精神飽滿地進行各項決策。

3.2 演算法

- 然而假使這些相同的步驟以稍微不同的順序來執行的話，如下：(1) 起床 (2) 脫掉睡衣 (3) 穿好衣服 (4) 洗澡 (5)吃早餐 (6)開車上班。
- 在這種情況下，我們的這位中級主管將會穿著溼的外衣去上班。
- 在電腦程式裡指定敘述式執行的順序稱為程式控制 (program control)。

3.3 虛擬程式碼

- 虛擬程式碼 (Pseudocode)是一種給人看的非正規的語言，用來幫助你發展演算法。
- 虛擬程式碼十分類似於日常生活上所用的英文，雖然它不是真的電腦程式語言，不過它卻是方便且用起來稱手的。
- 虛擬程式碼所寫的程式並不能在實際的電腦上執行。
- 它們是用來幫助你在真正以電腦程式語言（如C）撰寫程式前，「思考」這個程式該如何撰寫。
- 虛擬程式碼完全由字元所組成，因此你可以利用文書編輯軟體輕易地將之鍵入電腦。

3.3 虛擬程式碼

- 一份仔細設計過的虛擬程式碼的程式，可以很快地轉換成相對應的C程式。
- 虛擬程式碼只包含了動作敘述式－－就是那些當程式由虛擬碼轉換成C的時候被執行的部分。
- 宣告部份並不是可執行的敘述式。
- 他們是給編譯器看的訊息。

3.3 虛擬程式碼

- 例如，以下的定義
  - `int i;`
- 只是用來告知編譯器變數i的型別，並命令編譯器為此變數空出記憶體空間。
- 但此宣告在執行程式的時候，並不會引起任何的動作，例如輸入、輸出、或計算。
- 有些程式設計師會在虛擬程式的開頭，列出所有的變數並簡單地說明他們的用途。

3.4 控制結構

- 一般說來，程式中的敘述式是以他們在程式中的順序一個接一個地被執行。
- 這叫做循序式的執行 (sequential execution)。
- 不過我們將很快看到，有些C敘述式能夠讓你指定下一個執行的敘述式 (非循序式的)。
- 這叫做控制權的移轉 (transfer of control)。
- 在1960年代，人們發現任意使用控制權轉移，將會使得軟體的發展愈來愈困難。

3.4 控制結構

- 批評的焦點都集中在goto敘述式 (goto statement)的身上，因為它可讓程式設計師指定控制權轉移到程式中許多可能的地方。
- 於是「消除goto (goto elimination)」幾乎成了結構化程式設計的同義詞。
- 研究已經證明了程式可以不必使用goto敘述式來撰寫。
- 於是程式設計師所必須接受的挑戰，便是要將他們的設計風格轉變成〝沒有goto的程式設計方式〞。

3.4 控制結構

- 這個結果令人深刻印象的，如同軟體研發團隊減少了發展軟體的時間一樣，軟體計劃更常準時完成，除錯的時間也能夠更短。
- 使用結構化技術產生的程式會比較清楚，比較容易除錯以及修改，以及不容易產生錯誤。
- 研究告訴我們，所有的程式均可由三種*控制結構*寫成，他們是循序結構(sequence structure)，選擇結構和重複結構。

3.4 控制結構

- 循序結構是C內建的特性。
- 除非改變程式執行的流程，否則電腦會自動地按照你所寫的C敘述式的順序，一行一行地執行。
- 圖3.1所示的流程圖 (flowchart)片段，說明了C的循序結構。
- 流程圖是整個演算法或是演算法的一部分的圖形表示法。
- 流程圖使用具有特殊涵義的標誌來繪製，像是矩形，菱形，橢圓形以及圓形等等；這些標誌用稱為流向 (flowline) 的箭頭連接起來。

3.4 控制結構

- 如同虛擬程式碼一般，流程圖對發展和表示演算法非常地有幫助。雖然大多數的程式設計師較喜歡使用虛擬程式碼。
- 讓我們來看看圖3.1循序結構的流程圖片段。
- 我們用矩形 (rectangle symbol)，矩形也稱為動作符號 (action symbol) 來表示任何形式的動作，包括了計算或輸入／輸出的操作等。
- 圖上的流向代表動作執行的順序－先將grade加到total，然後再將counter加1。
- C允許我們在一個循序結構中放入任意個數的動作。

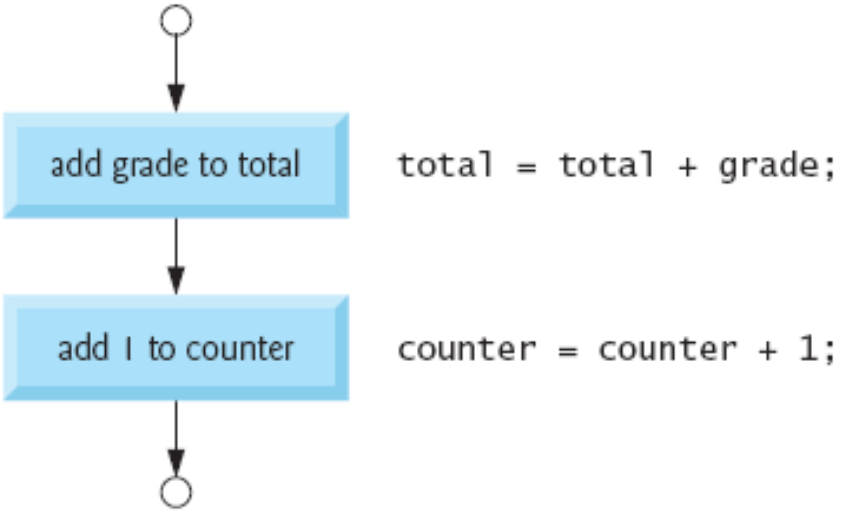

圖 3.1　C 循環結構的流程圖

3.4 控制結構

- 當我們畫流程圖表示完整的演算法時，要用兩個橢圓形符號 (oval symbol)，一個寫上"Begin"表示此流程圖的第一個符號，一個寫上"End"表示最後一個符號。
- 而當我們畫流程圖的片段(如圖3.1)時，則可省略橢圓形而以小圓形 (small circle symbols) 代替，此小圓形符號稱為連接符號 (connector symbols)。
- 流程圖中最重要的也許是菱形符號 (diamond symbol)，它也被稱為判斷符號 (decision symbol)。這種符號表示某項判斷將在此進行。

3.4 控制結構

- C語言以敘述式的形式提供了三種選擇結構。
- `if`選擇結構（3.5節）在條件式為真時執行 (選擇) 某項動作，而當條件為偽時則跳過這項動作。
- `if...else`選擇敘述式（3.6節）在條件為真時執行某項動作，而當條件為偽時則執行另一項動作。
- `switch`選擇敘述式（在第四章當中討論）會依運算式的不同，選擇執行許多動作中的一項。

3.4 控制結構

- `if`敘述式也稱為單一選擇敘述式 (single-selection statement)，因它只選擇或跳過一項動作。
- `if...else`敘述式也稱為雙重選擇敘述式 (double-selection statement)，因它會在兩種不同的動作之中選擇。
- `switch`敘述式稱為多重選擇敘述式 (multiple-selection statement)，因為它可以在許多不同的動作中選擇。
- C以敘述式的形式提供了三種重複結構，分別是`while` (3.7節)，`do...while`和`for` (兩者都在第四章討論)。
- 以上所介紹的便是C的所有控制結構。

3.4 控制結構

- C只有七種控制敘述式：循序、三種選擇和三種重複。
- 每個C程式都是依據其演算法的需要，組合這七種結構所形成的。
- 我們將可看到，每一種控制敘述式都會和圖3.1的循序結構一樣具有兩個小圓形，一個在控制敘述式的入口而另外一個在出口。
- 這種單一入口／單一出口的控制敘述式 (single-entry/single-exit control statements)，將可使程式的建構更為容易。

3.4 控制結構

- 我們可將一個控制結構的入口連接到另一控制結構的出口，來結合不同的控制敘述式。
- 這個動作十分類似小朋友玩的堆疊積木遊戲，因此我們稱之為控制敘述式的堆疊 (control-statement stacking)。
- 除了這種連接方式之外，只有另外一種方式可以連接控制敘述式，稱為巢狀的控制敘述式 (control-statement nesting)。
- 所以，任何C程式的建構，都可以用兩種連接方式連接七種控制結構來達成。
- 這就是簡單化的原理。

3.5 if 選擇敘述式

- 選擇結構可用來選取不同功能的動作。
- 例如，假設考試中及格的成績為60分，
- 以下的虛擬碼敘述式
  - If student's grade is greater than or equal to 60
    Print "Passed"
- 會判斷條件"student's grade is greater than or equal to 60"是真或偽。
- 若為真的話則印出"Passed"，然後繼續"執行"下一個虛擬碼敘述式（要記得虛擬碼不是真的程式語言）。

3.5 if 選擇敘述式

- 如果條件式為偽，則列印的動作不會執行，然後執行下一個虛擬碼敘述式。
- 請注意此選擇結構的第二行是縮排。
- 雖然這種縮排並非強制性的，但它卻有助於突顯結構化程式設計中的繼承結構。
- 另外，C編譯器會忽略像是空格、水平定位點以及新行等空白字元 (white-space characters)。

良好的程式設計習慣 3.1

在應當縮排的地方使用前後一致的縮排方式，將可大大地增進程式的可讀性。我們建議您，每個縮排的大小最好是 1/4 英吋的 tab，或是 3 個空格。在本書當中的縮排是使用 3 個空格。

3.5 if 選擇敘述式

- 上述虛擬碼If敘述式可改寫成如下的C程式
  - ```
    if ( grade >= 60 ) {
       printf( "Passed\n" );
    } /* end if */
    ```
- 我們可看到這個C程式十分地類似上述的虛擬碼。
    ```

良好的程式設計習慣 3.2

虛擬碼通常用在程式設計的「思考」過程中，然後我們便可將虛擬碼程式轉換爲 C 程式。

3.5 if 選擇敘述式

- 圖3.2的流程圖所示為單一選擇的if敘述式。
- 這張流程圖裡包含了最重要的流程符號－菱形符號（diamond symbol），菱形也稱為判斷符號(decision symbol)，它表示正在進行某項判斷。
- 判斷符號包含了一個運算式，例如一個控制條件，它可能為真或偽。

3.5 if 選擇敘述式

- 判斷符號會發出兩條流向。
- 一條代表條件式為真時的流向，另一條則代表條件式為偽時的流向。
- 判斷的進行可依據含有關係或相等運算子的條件式來進行。
- 事實上任何運算式皆可做為判斷的依據，當運算式的運算結果為零時便代表偽，而當運算結果為非零值時則代表真。

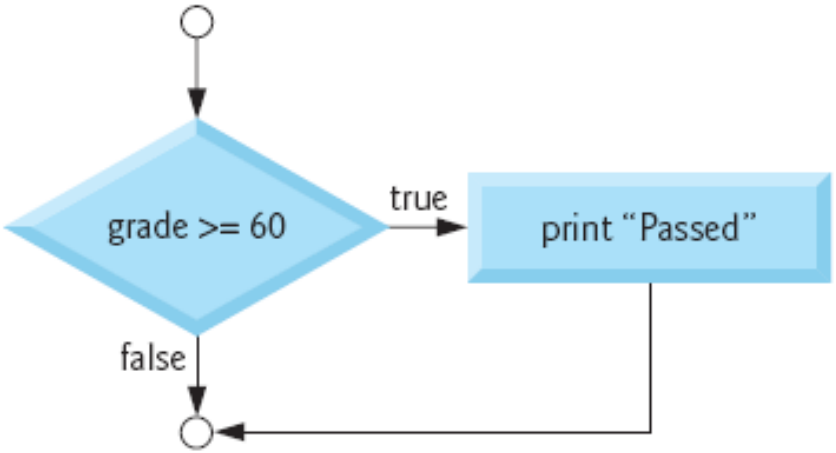

圖 3.2　單一選擇 `if` 敘述式的流程圖

3.5 if 選擇敘述式

- if結構也是個單一入口／單一出口的結構。
- 我們可想成有七個箱子，每個箱子中放著這七種控制敘述式中的一種。
- 這些在箱子裡的控制敘述式一開始時都是空無一物的。他們的矩形和菱形裡都沒寫上任何的文字。
- 而你的任務便是依照演算法的需要，以兩種方式（堆疊或巢狀）將這些控制結構組成程式。

3.6 if...else 選擇敘述式

- if...else選擇敘述式則讓你可依據條件的真偽，來執行不同的動作。
- 舉例來說，下列的虛擬碼敘述式
  - If student's grade is greater than or equal to 60
    Print "Passed"
    else
    Print "Failed"
- 會在學生的成績大於等於60的時候印出Passed，而在小於60的時候印出Failed。
- 在這種情況下，當列印完成後接下來的虛擬碼敘述式都將會被執行。請注意else的本體也是縮排的。

良好的程式設計習慣 3.3

請將 `if…else` 敘述式中所有的本體敘述式縮排。

良好的程式設計習慣 3.4

假如程式裡有數個層級的縮排，那麼每一個層級應該縮進的格數必須相同。

3.6 if...else 選擇敘述式

- 上述虛擬碼If...else敘述式可改寫成如下的C程式
  - ```
    if ( grade >= 60 ) {
       printf( "Passed\n" );
    } /* end if */
    else {
       printf( "Failed\n" );
    } /* end else */
    ```
- 圖3.3的流程圖完善地表示了if...else結構的控制流程。
- 同樣的，除了小圓形和箭號之外，這張流程圖中也只包含了矩形 (動作) 和菱形 (判斷)。
    ```

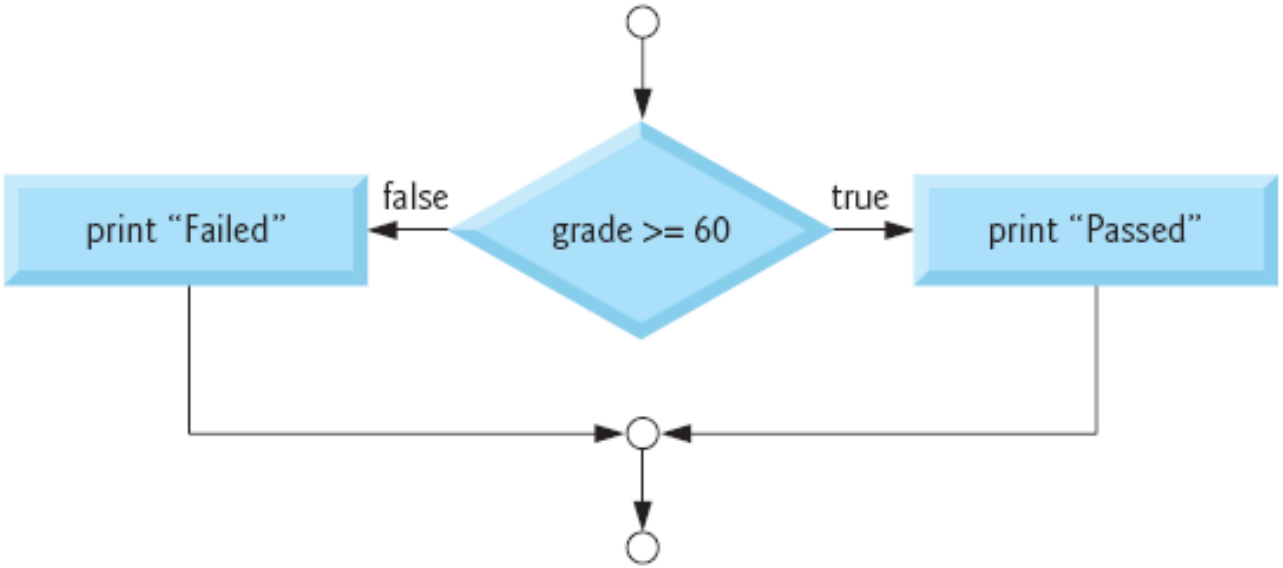

圖 3.3　C 之雙重選擇 `if...else` 敘述式的流程圖

3.6 if...else 選擇敘述式

- C提供了與if...else敘述式十分類似的條件運算子(?:)。
- 條件運算子是C中唯一的三元運算子 (ternary operator)－它使用了三個運算元。
- 這些運算元加上條件運算子構成了條件運算式(conditional expression)。
- 其中第一個運算元是條件。
- 第二個運算元是當條件為真時整個條件運算式的值，第三個運算元則是當條件為偽時整個條件運算式的值。

3.6 if...else 選擇敘述式

- 例如下面的printf敘述式：
  - ```
    printf( "%s\n", grade >= 60 ?
       "Passed" : "Failed" );
    ```
- 包含了一個條件運算式，當其條件grade>=60為真時，值為字串"Passed"，而當其條件為偽時，值為字串"Failed"。
- printf的格式控制字串裡含有一個轉換指定詞%s，用來印出一列字串。
- 因此這個printf敘述式，其效用與上述的if...else敘述式相同。
    ```

3.6 if...else 選擇敘述式

- 條件運算式裡的第二和第三個運算元也可以是某項要執行的動作。
- 例如，底下的運算式
  - ```
    grade >= 60 ? printf( "Passed\n" ) :
        printf( "Failed\n" );
    ```
- 將被讀成「如果grade大於等於60，就執行printf("Passed\n")，否則執行printf("Failed\n")」。這個敘述式的效用也和前述的if...else敘述式相同。
- 而我們也將可看到，條件運算子能用在if...else敘述式無法適用的情況。
    ```

3.6 if...else 選擇敘述式

- 我們可將if...else敘述式放到另一個if...else敘述式裡，構成巢狀的if...else敘述式，用以檢測多重的狀況。
- 舉例來說，下列的虛擬碼敘述式會在考試成績大於等於90的時候印出A，大於等於80時印出B，大於等於70時印出C，大於等於60時印出D，而其他的成績則印出F。
  - If student's grade is greater than or equal to 90
    Print "A"
    else
    If student's grade is greater than or equal to 80
    Print "B"
    else
    If student's grade is greater than or equal to 70
    Print "C"
    else
    If student's grade is greater than or equal to 60
    Print "D"
    else
    Print "F"

3.6 if…else 選擇敘述式

- 這段虛擬碼可寫成 C 如下

```
if ( grade >= 90 )
   printf( "A\n" );
else
   if ( grade >= 80 )
      printf("B\n");
   else
      if ( grade >= 70 )
         printf("C\n");
      else
         if ( grade >= 60 )
            printf( "D\n" );
         else
            printf( "F\n" );
```

3.6 if...else 選擇敘述式

- 如果變數grade大於等於90的話，前四個條件都將為真，但只有第一個測試的printf敘述式會執行。
- 在這個printf執行之後，最外層之if...else敘述式的else部分將會被跳過。

3.6 if…else 選擇敘述式

- 許多的C程式設計師喜歡將上述的if敘述式寫成

```
if ( grade >= 90 )
   printf( "A\n" );
else if ( grade >= 80 )
   printf( "B\n" );
else if ( grade >= 70 )
   printf( "C\n" );
else if ( grade >= 60 )
   printf( "D\n" );
else
   printf( "F\n" );
```

3.6 if...else 選擇敘述式

- 這兩種方式對C編譯器來說是相同的。
- 而後者較受到歡迎的原因是它可避免因過深的縮排導致程式向右傾斜。
- if選擇敘述式認為它的本體中只有一個敘述式，
- 因此若我們想在 if 結構中放入數個敘述式，便必須用大括號（{ 和 }）將它們包起來。
- 這些被包在大括號裡的敘述式稱為複合敘述式 (compound statement) 或是一個區塊 (block)。

軟體工程的觀點 3.1

複合敘述式可放在程式中任何可放單一敘述式的地方。

3.6 if...else 選擇敘述式

- 下面的例子裡在if...else敘述式的else部分中，包含了一個複合敘述式。

```
if ( grade >= 60 ) {
   printf( "Passed.\n" );
} /* end if */
else {
   printf( "Failed.\n" );
   printf( "You must take this course
again.\n" );
} /* end else */
```

3.6 if...else 選擇敘述式

- 在這個例子裡，如果grade小於60的話，程式將會執行else本體內的兩道printf敘述式，印出
  - ```
    Failed.
    You must take this course again.
    ```
- 請注意包住else子句的兩個敘述式的大括號。
- 他們非常的重要。如果沒寫這兩個括號的話，第二個printf敘述式

```
printf( "You must take this course again.\n" );
```

- 將會不屬於if...else敘述式，亦即不論grade是否小於60，這個敘述式都會被執行。
```

常見的程式設計錯誤 3.1

忘了用大括號將複合敘述式包起來。

3.6 if...else 選擇敘述式

- 語法錯誤會在編譯時產生。
- 邏輯錯誤會在執行時期造成影響。
- 致命的邏輯錯誤會使得程式失敗並提早終止。
- 非致命的邏輯錯誤則可讓程式繼續執行，但會產生不正確的執行結果。

軟體工程的觀點 3.2

就像複合敘述式可放在任何單一敘述式可放的地方一樣，複合敘述式內也可以不含任何的敘述式，即空的敘述式。空敘述式的表示方式是在可放敘述式的地方放一個分號 (;)。

常見的程式設計錯誤 3.2

在 `if` 敘述式的條件式之後放置一個分號，像是「`if (grade >= 60);`」將會使單一選擇的 `if` 敘述式產生邏輯錯誤，而使雙重選擇的 `if` 敘述式產生語法錯誤。

測試和除錯的小技巧 3.1

在輸入個別的敘述式之前先輸入左右大括號，可以避免忘記輸入一個或兩個大括號，而造成語法錯誤，或是邏輯錯誤。

3.7 while 重複敘述式

- 重複敘述式 (repetition statement) 可讓你指定在某種條件持續為真時，重複執行同一項動作。
- 下面這個虛擬碼敘述式
  - While there are more items on my shopping list
    Purchase next item and cross it off my list
- 描述了在購物行程中的重複動作。
- 其中的條件"there are more items on my shopping list"可能是真也可能是偽。
- 如果為真的話，那麼動作"Purchase next item and cross it off my list"就會被執行。
- 而只要此條件持續為真，這項動作就會重複地執行。

3.7 while 重複敘述式

- While重複敘述式內的敘述式構成了while的本體。
- while敘述式的本體可以是單一的敘述式，也可以是複合敘述式。
- 當條件變成偽時（當購買了shopping list中的最後一項並將之刪除後），
- 重複動作便停止，而接下來執行的是重複結構之後的第一個虛擬碼敘述式。

常見的程式設計錯誤 3.3

在 while 結構的本體內，沒有任何一個動作能讓 while 的條件變成僞。通常這種重複結構將不會停止－－此種錯誤稱爲「無窮迴圈」。

常見的程式設計錯誤 3.4

將關鍵字 `while` 拼成具有大寫 `W` 的 `While` (請不要忘記 C 是種大小寫有所區分的語言)。C 的所有關鍵字，如 `while`、`if` 和 `else` 等，都是由小寫字母所組成的。

3.7 while 重複敘述式

- 讓我們來看看下面的while例子，此程式片段是用來找出第一個大於100的3的次方數。
- 假設整數變數product已被設定初值為3。
- 當下列的while重複敘述式執行完畢時，product應該就是我們所要的答案。

```
product = 3;
while ( product <= 100 ) {
    product = 3 * product;
} /* end while */
```

- 圖3.4的流程圖表示了這個while重複敘述式的控制流程。

3.7 while 重複敘述式

- 同樣的，我們看到在此圖中除了小圖形和箭號之外，只有一個矩形和一個菱形。
- 流程圖清楚地顯示了重複性。
- 從矩形流出的流向線會在每次迴圈時折回到原來判斷步驟的開始處做測試，直到判斷最終變成偽時才結束。
- 此時會離開while敘述式，而控制權將傳給程式中的下一個敘述式。

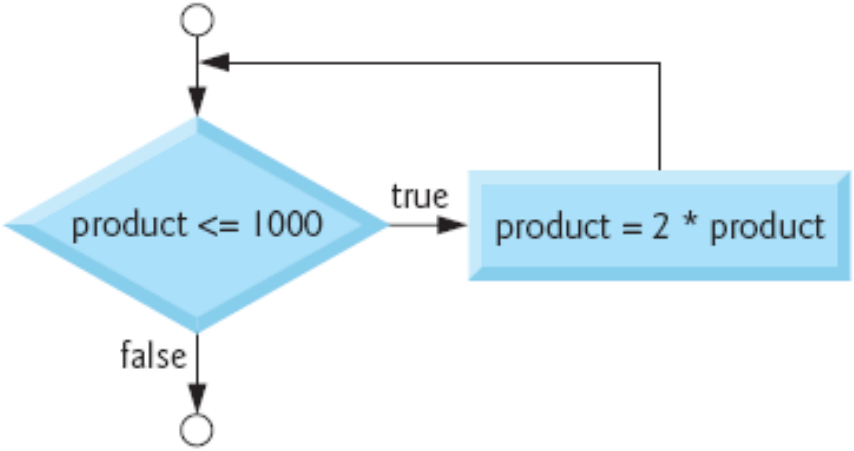

圖 3.4　`while` 重複敘述式的流程圖

3.7 while 重複敘述式

- 一開始進入while敘述式時，product的值為3。
- 之後，變數product將被重複地乘以3，其值將變成9,27以及81。
- 當product變為243後，while敘述式內的條件product<=100便變為偽。
- 此時重複動作將結束，而product最終的值為243。
- 程式便從while之後的第一個敘述式繼續執行下去。

3.8 建構演算法案例研究 1：計數器控制的重複結構

- 為了說明如何發展一個演算法，我們利用了數種方法來解決計算全班平均成績的問題。
- 請看下列的問題描述：
  - 十個學生的一個班級進行一次測驗。你手上已有這次測驗的成績（從0到100的整數）。請求出此班的平均成績為何。
- 全班平均即等於所有成績的總和除以生的人數。
- 在電腦上解決此問題的演算法必須輸入每個學生的成績，執行求平均值的計算，然後再將結果印出來。

3.8 建構演算法案例研究 1：計數器控制的重複結構

- 讓我們先用虛擬碼列出所有需執行的動作，並指定這些動作的執行順序。
- 我們利用計數器控制的重複結構 (counter-controlled repetition)，一次一個地輸入這些成績。
- 在這項技巧裡，我們用了一個稱為counter (計數器)的變數，來指定某一組陳述句應被執行的次數。
- 在本例中，當counter超過10時，重複動作便告結束。

3.8 建構演算法案例研究 1：計數器控制的重複結構

- 本節我們將只列出虛擬碼演算法 (圖3.5)及其相對應的C程式 (圖3.6)。
- 下一節再詳細介紹如何發展虛擬碼演算法。
- 計數器控制的重複通常也稱為明確的重複，因為重複的次數在迴圈開始執行之前便已得知。

```
1   Set total to zero
2   Set grade counter to one
3
4   While grade counter is less than or equal to ten
5       Input the next grade
6       Add the grade into the total
7       Add one to the grade counter
8
9   Set the class average to the total divided by ten
10  Print the class average
```

圖 3.5　利用計數器控制的重複結構來解決全班平均問題的虛擬碼演算法

```c
/* Fig. 3.6: fig03_06.c
   Class average program with counter-controlled repetition */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int counter; /* number of grade to be entered next */
   int grade; /* grade value */
   int total; /* sum of grades input by user */
   int average; /* average of grades */

   /* initialization phase */
   total = 0; /* initialize total */
   counter = 1; /* initialize loop counter */

   /* processing phase */
   while ( counter <= 10 ) { /* loop 10 times */
      printf( "Enter grade: " ); /* prompt for input */
      scanf( "%d", &grade ); /* read grade from user */
      total = total + grade; /* add grade to total */
      counter = counter + 1; /* increment counter */
   } /* end while */
```

圖 3.6　以計數器控制重複結構來解決全班平均問題的 C 程式及其執行範例

```
24
25      /* termination phase */
26      average = total / 10; /* integer division */
27
28      printf( "Class average is %d\n", average ); /* display result */
29      return 0; /* indicate program ended successfully */
30   } /* end function main */
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

圖 3.6　以計數器控制重複結構來解決全班平均問題的 C 程式及其執行範例

3.8 建構演算法案例研究 1：計數器控制的重複結構

- 請注意在上述的演算法裡指到 total 和 counter 的參考。
- 其中total是個變數用來累計一連串數值的總和。
- counter也是個變數，用來計數－在此例中計算成績輸入的個數。

3.8 建構演算法案例研究 1：計數器控制的重複結構

- 變數total應在被使用之前清為0，否則算出來的總和會包括先前存在total記憶體位置的值。
- 變數counter則通常設定初值為0或1，這點需視其使用情形而定 (我們將會展示這兩種使用情形的範例)。
- 一個沒有初始化的變數包含「垃圾」值(“garbage ” value)－意指保留在該記憶體位置的其他變數的值。

常見的程式設計錯誤 3.5

如果 counter 或 total 沒有設定初值，這樣子程式執行的結果將可能不正確。這是個邏輯錯誤的例子。

測試和除錯的小技巧 3.2

必須初始化所有的 counter 及 total。

3.8 建構演算法案例研究 1：計數器控制的重複結構

- 請注意到在本程式中計算的平均數會產生一個整數值 81。
- 而事實上本例的成績總和為817，除以10之後應該是81.7－個具有小數點的數。
- 我們將在下一節介紹如何處理這種數 (稱為浮點數)。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 現在讓我們將全班平均問題一般化。
- 請看下面所述的問題：
  - 發展一個求全班平均的程式，它可以處理任何數目的成績。
- 在第一個全班平均的例子裡，成績的個數（10個）是事先知道的。
- 此程式必須能夠處理任何數目的成績。
- 那麼這個程式如何判斷何時該停止成績的輸入呢？它又如何知道何時開始計算及印出班平均值呢？

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 解決此問題的方法之一是利用一種稱警示值 (sentinel value) (也稱為信號值 signal value、虛值 dummy value、或旗標值 flag value) 的特殊數值，來代表「資料輸入的結束」。
- 使用者持續地鍵入成績，直到所有的成績都輸入為止。
- 接著便鍵入警示值，以表示最後一個成績已經輸入完畢。
- 警示控制的重複結構通常也稱為不確定次數的重複結構 (indefinite repetition)，因為在迴圈開始執行之前並不知道重複的次數。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 很顯然的，警示值的選擇必須不能與可接受的輸入值混淆。
- 由於測驗的成績通常都是非負的整數，因此我們可用-1來做為這個問題的警示值。
- 如此一來，執行這個全班平均程式將可以處理一連串如95, 96, 75, 74, 89和-1的輸入。
- 程式接下來便計算並印出95, 96, 75, 74, 89的全班平均值 (-1是警示值，所以不能把它放到平均數的計算裡)。

常見的程式設計錯誤 3.6

所選擇的警示值也是個可能的輸入資料值。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 我們用一種從上而下逐步改進的技術 (top-down, stepwise refinement)，來發展上述的全班平均程式。這種技術特別適用於結構良好之程式的開發。
- 我們先從代表總敘述式的虛擬碼開始：
  - 請求出此班的平均成績為何。
- 總敘述式是一個單一敘述式，它涵蓋了程式所有的功能。
- 因此，總敘述式是程式的完整表示。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 不過，總敘述式所涵蓋的資訊實在是太粗略了，我們並不能以它來撰寫C程式。
- 因此我們現在開始進行改進的程序。
- 我們先將總敘述式分割成數項較小的工作，並依它們執行的順序列出來。
- 於是我們得到了下列初步的改進 (first refinement)。
  - Initialize variables
    Input, sum, and count the quiz grades
    Calculate and print the class average
- 到目前為止我們只使用了循序結構－以上所列的各步驟是一個接一個執行。

軟體工程的觀點 3.3

總敘述式及每一次的改進都是一份完整的演算法，不同的只是詳細的程度。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 在第二個改進 (second refinement) 前，先讓我們來看看會用到哪些變數。
- 我們需要紀錄所有數值的總和，計算有多少個數值已被處理，接收輸入成績的變數，以及存放計算出來平均值的變數。
- 下面這個虛擬碼敘述式
  - Initialize variables
- 可改進成如下：
  - Initialize total to zero
    Initialize counter to zero

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 請注意，只有total和counter需被設定初值；變數average(算出來的平均值)和grade(使用者輸入的成績)則不需設定初值，因為這兩個變數的值會被讀入(見第二章)的值所實際取代。
- 下面的虛擬碼敘述式
  - Input, sum, and count the quiz grades
- 需使用重複結構 (即迴圈) 來連續輸入每一個成績。
- 由於事先並不知道將要處理多少個成績，因此我們選擇使用警示值控制重複結構。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 使用者將一次輸入一個成績。
- 在最後一個成績輸入之後，使用者接著輸入警示值。
- 當每個成績輸入時，程式都必須檢查這個值。如果檢查到警示值已輸入的話，迴圈便將終止。
- 於是上述的虛擬碼敘述式便可改進為
  - Input the first grade
    While the user has not as yet entered the sentinel
        Add this grade into the running total
        Add one to the grade counter
        Input the next grade (possibly the sentinel)

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 請注意在此虛擬碼中，我們並沒有將while敘述式本體的敘述式用大括號包起來。
- 我們只是將他們縮排，以表示他們是屬於while的。
- 再次強調，虛擬程式碼只是用來幫助發展程式。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 下面這個虛擬碼敘述式
  - Calculate and print the class average
- 可改進成如下：
  - If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
    else
    Print “No grades were entered”
- 請注意，我們在這裡小心地檢查了除以零的可能性。除以零是一種致命錯誤 (fatal error)，它將會使程式當掉 (bombing，crashing)。
- 完整的第二次改進可見圖3.7。

常見的程式設計錯誤 3.7

嘗試除以零會造成一個致命錯誤。

良好的程式設計習慣 3.5

當執行除法時，若除數爲一值可能爲 0 的運算式，那麼請在程式裡檢查這項可能性，並在檢查到爲 0 時進行一些妥善的處理 (如印出錯誤訊息)，而不要讓致命錯誤眞的發生。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 在圖3.5和圖3.7裡，我們在虛擬碼中加了一些空行以增進可讀性。
- 事實上，空行也將程式區隔成數個不同的階段。

```
1   Initialize total to zero
2   Initialize counter to zero
3
4   Input the first grade
5   While the user has not as yet entered the sentinel
6       Add this grade into the running total
7       Add one to the grade counter
8       Input the next grade (possibly the sentinel)
9
10  If the counter is not equal to zero
11      Set the average to the total divided by the counter
12      Print the average
13  else
14      Print "No grades were entered"
```

圖 3.7　利用警示訊號控制的重複結構來解決全班平均問題的虛擬碼演算法

軟體工程的觀點 3.4

大多數的程式在邏輯上都可分成三個階段：初始化階段爲程式裡的變數設定初值；處理階段負責輸入資料並修改相對應的變數；以及結束階段，負責計算並印出最後的結果。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 圖3.7的虛擬碼演算法解決了本節所述的問題。
- 這個演算法在經過兩個階層的改進之後發展完成。
- 有時候可能會需要較多階層的改進。

軟體工程的觀點 3.5

當虛擬碼演算法已詳細地足以轉換成 C 程式時，你便可停止從上而下逐步改進的程序。此時要撰寫 C 程式通常已經是很明確了。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 圖3.8所示為此演算法所對應的C程式及執行範例。
- 雖然我們所輸入的成績都是整數，不過其平均值卻是具有小數點的小數。
- `int`型別並無法表示這種數。
- 因此這個程式使用了`float`資料型別來處理小數（稱為浮點數，floating-point numbers），並使用一個稱為強制型別轉換運算子(cast operator)的特殊運算子，來處理平均值的計算。
- 我們將會在稍後繼續介紹這些功能。

```c
/* Fig. 3.8: fig03_08.c
   Class average program with sentinel-controlled repetition */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int counter; /* number of grades entered */
   int grade; /* grade value */
   int total; /* sum of grades */

   float average; /* number with decimal point for average */

   /* initialization phase */
   total = 0; /* initialize total */
   counter = 0; /* initialize loop counter */

   /* processing phase */
   /* get first grade from user */
   printf( "Enter grade, -1 to end: " ); /* prompt for input */
   scanf( "%d", &grade ); /* read grade from user */

```

圖 3.8　以警示值控制重複結構來解決全班平均問題的 C 程式及其執行範例

```
23      /* loop while sentinel value not yet read from user */
24      while ( grade != -1 ) {
25         total = total + grade; /* add grade to total */
26         counter = counter + 1; /* increment counter */
27
28         /* get next grade from user */
29         printf( "Enter grade, -1 to end: " ); /* prompt for input */
30         scanf("%d", &grade); /* read next grade */
31      } /* end while */
32
33      /* termination phase */
34      /* if user entered at least one grade */
35      if ( counter != 0 ) {
36
37         /* calculate average of all grades entered */
38         average = ( float ) total / counter; /* avoid truncation */
39
40         /* display average with two digits of precision */
41         printf( "Class average is %.2f\n", average );
42      } /* end if */
43      else { /* if no grades were entered, output message */
44         printf( "No grades were entered\n" );
45      } /* end else */
```

圖 3.8　以警示值控制重複結構來解決全班平均問題的 C 程式及其執行範例

```
46
47      return 0; /* indicate program ended successfully */
48   } /* end function main */
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

```
Enter grade, -1 to end: -1
No grades were entered
```

圖 3.8　以警示值控制重複結構來解決全班平均問題的 C 程式及其執行範例

3.9 以從上而下逐步改進的方式建構演算法 案例研究 2：警示訊號控制的重複結構

- 注意圖3.8程式之while 迴圈內(第24行)的複合敘述式，再一次強調，必須用大括號將迴圈內執行的四個敘述式括起來。
- 若沒有這兩個括號的話，迴圈本體內的後面三個敘述式將被視為在迴圈之外，電腦將會誤認這些敘述式如下：

```
while ( grade != -1 )
   total = total + grade; /* add grade to total */
counter = counter + 1; /* increment counter */
printf( "Enter grade, -1 to end: " ); /* prompt for
input */
scanf( "%d", &grade ); /* read next grade */
```

假使使用者第一次輸入的grade不等於-1的話，這樣子的程式將會造成無窮迴圈。

良好的程式設計習慣 3.6

在警示值控制迴圈裡，要求輸入資料的提示訊息應明確地告訴使用者警示值爲何。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 計算出來的平均成績並不一定是整數。
- 通常平均數會是如7.2或-93.5之類的小數。
- 這種數值稱為浮點數，以資料型別`float`來表示。
- 變數average被宣告成`float`型別 (第12行)，以便存放小數。
- 不過`total/counter`的計算結果是一個整數，這是因為`total`和`counter`這兩個變數都是整數之故。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 在C裡，兩個整數相除 (integer division) 的小數部分將被捨棄。
- 因此當這個相除的運算執行後，小數部分已被捨棄，此時被指定給average的值將只剩下整數部分。
- 為了能保留計算結果的小數部分，我們必須製造一個暫時的浮點數。
- C提供了單元強制型別轉換運算子 (cast operator) 來負責這項工作。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 第38行
  - `average = ( float ) total / counter;`
- 含有 (float) 這個強制型別轉換運算子，它會為它的運算元total產生一個暫時的浮點數拷貝。
- 而存放在total的值仍然是個整數。
- 以這種方式來使用強制型別轉換運算子稱為明確地轉換 (explicit conversion)。
- 此時，這項運算變成了一個浮點數（total的暫時float版本）除以一個整數（值存於counter）。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 大多數的電腦只能夠執行運算元型別相同的運算式。
- 為了確保所有的運算元型別相同，編譯器會對某些運算元執行一種稱為提升 (promotion) 的動作（也稱為隱含式轉換，implicit conversion）。
- 例如在一個含有`int`和`float`型別的運算式裡，ANSI標準規定對`int`的運算元複製，並將之提升為`float`。
- 在我們的例子裡，`counter`被複製之後便被提升為`float`，然後執行兩個浮點數的相除，再將相除結果指定給`average`。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- ANSI標準提供了一套關於不同型別之運算元提升的規則。
- 我們將在第五章裡討論標準資料型別以及它們的提升規則。
- 強制型別轉換運算子（即cast運算子）可適用於任何的資料型別。
- 這種運算子的表示法是括號括住資料型別的名稱。
- cast運算子是單元運算子 (unary operator)，亦即只有一個運算元的運算子。
- C也支援一元的加(+)和減(-)運算子，所以你可撰寫像-7或+5之類的運算式。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- cast運算子的結合性是由右至左，其運算優先順序與其他的一元運算子 (如一元的+和一元的-) 相同。
- 這種等級的優先順序要比*、/和%等**乘法運算子 (multiplicative operators)** 運算子高一級，而比括號低一級。
- 圖3.8的程式使用`printf`的轉換指定詞`%.2f` (第41行) 來印出average的值。
- 其中f表示將會有個浮點數要被列印，
- 而 .2則指定了此值被列印時的精準度 (precision)。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 如果我們用的是%f這個轉換指定詞（沒有指定精準度），那麼列印時將使用預設精準度 (default precision) 6－就如同我們使用%.6f這樣的轉換指定詞。
- 當列印浮點數時，其值會被四捨五入 (rounded) 成所指定的精準度。
- 在記憶體內的值未被改變。
- 如下兩個敘述式執行後，將會印出3.45和3.4。

```
printf( "%.2f\n", 3.446 ); /* prints 3.45 */
printf( "%.1f\n", 3.446 ); /* prints 3.4  */
```

常見的程式設計錯誤 3.8

在 scanf 敘述式之格式控制字串內使用具有精準度的轉換指定詞是不正確的。精準度只能使用在 printf 的轉換指定上。

常見的程式設計錯誤 3.9

在使用浮點數時假設他們能夠完全精準地被表示出來，可能會導致錯誤的結果。大多數的電腦都只能近似地表示浮點數。

測試和除錯的小技巧 3.3

不要比較兩個浮點數的相等性。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 雖然浮點數並不一定100%的精確，但他們還是可以應用在許多地方。
- 例如當我們說正常的體溫為98.6度（華氏度）時，我們並不計較精準到何種程度。
- 當我們從體溫計上讀出98.6度時，真正的值可能是98.5999473210643。
- 但將此值看成98.6卻是較實用的。

3.9 以從上而下逐步改進的方式建構演算法　案例研究 2：警示訊號控制的重複結構

- 相除也會產生浮點數。
- 當我們用10除以3時，得到的是一個無窮小數3.3333333...。
- 電腦只配置了固定大小的空間來存放這種數值，因此電腦所存的值只是個近似值。

3.10 以從上而下逐步改進的方式建構演算法　案例研究 3：巢狀的控制結構

- 讓我們來看看另一個完整的問題。
- 這次我們也將使用虛擬碼，以及從上而下逐步改進的方式來建構演算法，並撰寫出其相對應的C程式。
- 稍早我們曾討論過，控制結構可以堆疊在另一控制結構的上方，就如同小孩子堆積木一樣。
- 在本案例研討中，我們將可看到在C中連接控制敘述式唯一的另一種方式，即將一控制結構巢狀地(nesting)包含在另一控制敘述式之中。

3.10 以從上而下逐步改進的方式建構演算法 案例研究 3：巢狀的控制結構

- 請看下列的問題描述：
  - 某所學校開授一門課程，專門教授學生們有關房地產經紀人執照考試的知識。去年有數位學生修完這門課程，並參加了執照考試。很自然的，這所學校會想要知道她的學生們在這次考試的表現如何。假設你被授意寫個程式來統計考試結果。你手上有這10位學生的名單，以及他們的考試結果（1代表過關，2代表失敗）。

3.10 以從上而下逐步改進的方式建構演算法　案例研究 3：巢狀的控制結構

- 你的程式應按照下列的規定來統計此次考試的結果：
  - 輸入每個考試結果 (即1或2)。在程式每次要求輸入下一個考試結果時，提示"Enter result"這個訊息。
  - 計算每一種考試結果的個數。
  - 列出考試結果的統計，告知有多少個學生過關，有多少個學生失敗。如果超過8位學生通過這項考試的話，印出"Raise tuition"這個訊息。

3.10 以從上而下逐步改進的方式建構演算法　案例研究 3：巢狀的控制結構

- 在仔細地讀過問題的描述之後，我們做了下列的觀察報告：
  - 此程式必須處理10個考試結果。可用計數器控制式的迴圈。
  - 每一個考試結果是一個數字：不是1便是2。每次程式讀進考試成績時，必須判斷此數為1或2。在我們的演算法裡是判斷它是否為1，若不是1的話，則假設它是2（本章末的一道習題將探討如此的假設所帶來的影響）。
  - 會用到兩個計數器：一個計算過關的學生人數，一個計算失敗的學生人數。
  - 在程式處理完所有的考試結果後，它必須判斷是否有8位以上的學生通過這項考試。

3.10 以從上而下逐步改進的方式建構演算法　案例研究 3：巢狀的控制結構

- 現在讓我們來進行從上而下逐步改進的程序。
- 我們先從代表總敘述式的虛擬碼開始：
  - Analyze exam results and decide if instructor should receive a bonus
- 同樣的，我們必須強調總敘述式是此程式的完整表示敘述，但在虛擬碼能夠轉換成C程式之前，需要經過數個階段的改進。

3.10 以從上而下逐步改進的方式建構演算法　案例研究 3：巢狀的控制結構

- 我們的第一次改進如下：
  - Initialize variables
  - Input the ten quiz grades and count passes and failures
  - Print a summary of the exam results and decide if instructor should receive a bonus
- 雖然我們已完全地表示整個程式，不過更進一步的改進還是需要的。
- 現在來看看有哪些變數。
- 需要有兩個計數器紀錄過關和失敗的個數，一個控制迴圈的計數器，及一個存放使用者輸入值的變數。

3.10 以從上而下逐步改進的方式建構演算法　案例研究 3：巢狀的控制結構

- 以下的虛擬碼敘述式
  - Initialize variables
- 可改進成如下：
  - Initialize passes to zero
    Initialize failures to zero
    Initialize student to one
- 請注意我們只對計數器和學生人數設定初值。
- 下列的虛擬碼敘述式
  - Input the ten quiz grades and count passes and failures
- 需要一個迴圈來連續輸入每個考試結果。

3.10 以從上而下逐步改進的方式建構演算法　案例研究 3：巢狀的控制結構

- 因為在本例中已事先知道共有十個考試成績，所以採用計數器控制式的迴圈。
- 在迴圈裡 (巢狀地，nested) 需有一個雙重選擇結構來判斷每一個考試結果是過關或失敗，並遞增相對應的計數器。

3.10 以從上而下逐步改進的方式建構演算法　案例研究 3：巢狀的控制結構

- 於是上述的虛擬碼敘述式便可改進為
  - While student counter is less than or equal to ten
    Input the next exam result

    If the student passed
    Add one to passes
    else
    Add one to failures

    Add one to student counter
- 請注意我們用空行將 `if...else` 控制結構與其他部分區隔，用以增進程式的可讀性。

3.10 以從上而下逐步改進的方式建構演算法　案例研究 3：巢狀的控制結構

- 以下的虛擬碼敘述式
  - Print a summary of the exam results and decide if instructor should receive a bonus
- 可改進成如下：
  - Print the number of passes
    Print the number of failures
    If more than eight students passed
    　　Print “Bonus to instructor!”
- 完整的第二次改進列在圖3.9。
- 請注意在此圖中我們也用空行將`while`結構區隔開來，以增進程式的可讀性。

3.10 以從上而下逐步改進的方式建構演算法　案例研究 3：巢狀的控制結構

- 現在，這個虛擬碼程式已經改進得可以轉換成C程式了。
- 圖3.10列出了所轉成的C程式，及兩個執行範例。
- 我們利用了C的一項特性－－在宣告變數時順便為它設初值。
- 這種設定初值的動作是在編譯時進行。

```
1   Initialize passes to zero
2   Initialize failures to zero
3   Initialize student to one
4
5   While student counter is less than or equal to ten
6           Input the next exam result
7
8           If the student passed
9               Add one to passes
10          else
11              Add one to failures
12
13          Add one to student counter
14
15  Print the number of passes
16  Print the number of failures
17  If more than eight students passed
18      Print "Bonus to instructor!"
```

圖 3.9　有關考試結果問題的虛擬碼

增進效能的小技巧 3.1

在宣告變數時便爲他們設定初值，可減少程式的執行時間。

增進效能的小技巧 3.2

許多我們在這裡提到的「增進效能的小技巧」對效率的改進微不足道，因此讀者可能不想理睬它們。注意到把這些影響效能的原因累積起來，能使程式明顯增快。而在大量重複結構的迴圈當中做微小的效能改進時，就會對效能產生顯著的影響。

```c
/* Fig. 3.10: fig03_10.c
   Analysis of examination results */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   /* initialize variables in definitions */
   int passes = 0; /* number of passes */
   int failures = 0; /* number of failures */
   int student = 1; /* student counter */
   int result; /* one exam result */

   /* process 10 students using counter-controlled loop */
   while ( student <= 10 ) {

      /* prompt user for input and obtain value from user */
      printf( "Enter result ( 1=pass,2=fail ): " );
      scanf( "%d", &result );

      /* if result 1, increment passes */
      if ( result == 1 ) {
         passes = passes + 1;
```

圖 3.10　有關考試結果問題之 C 程式及執行範例

```
24          } /* end if */
25          else {/* otherwise, increment failures */
26             failures = failures + 1;
27          } /* end else */
28
29          student = student + 1; /* increment student counter */
30       } /* end while */
31
32       /* termination phase; display number of passes and failures */
33       printf( "Passed %d\n", passes );
34       printf( "Failed %d\n", failures );
35
36       /* if more than eight students passed, print "Bonus to instructor!" */
37       if ( passes > 8 ) {
38          printf( "Bonus to instructor!\n" );
39       } /* end if */
40
41       return 0; /* indicate program ended successfully */
42    } /* end function main */
```

圖 3.10　有關考試結果問題之 C 程式及執行範例

```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Passed 6
Failed 4
```

圖 3.10　有關考試結果問題之 C 程式及執行範例

```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Passed 9
Failed 1
Bonus to instructor!
```

圖 3.10　有關考試結果問題之 C 程式及執行範例

軟體工程的觀點 3.6

根據經驗顯示，利用電腦解決問題時，最困難的部分便是解決方法之演算法的開發。一旦有了正確的演算法，接下來便很容易寫出可以執行的 C 程式。

軟體工程的觀點 3.7

有些程式設計師沒使用程式開發工具 (如虛擬碼) 便開始撰寫程式。他們認爲最終的目的是在電腦上解決他們的問題，因此撰寫虛擬碼只會延遲程式開發的進度。

3.11 指定運算子

- C提供了數種指定運算子，使得指定運算式可以縮寫。
- 例如，以下的敘述式：
  - `c = c + 3;`
- 可利用加法指定運算子 += (addition assignment operator +=) 縮寫成
  - `c += 3;`
- +=運算子會把在此運算子右邊的運算式的值，加上此運算子左邊變數的值，然後將結果存到運算子左邊的變數。

3.11 指定運算子

- 任何如下面格式的敘述式
  - *variable = variable operator expression;*
- 其中的運算子(operator)是二元運算子+、-、*、/或%（第10章會再介紹其他的運算子）中的一種，他們都可以寫成下面的格式
  - *variable operator= expression;*
- 因此指定運算 c += 3 是將 c 加上 3。
- 圖3.11列出了算術指定運算子，使用這些運算子的運算式範例，以及展開式。

指定運算子	範例運算式	展開式	指定值
假設： `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
+=	`c += 7`	`c = c + 7`	10 到 c
-=	`d -= 4`	`d = d - 4`	1 到 d
*=	`e *= 5`	`e = e * 5`	20 到 e
/=	`f /= 3`	`f = f / 3`	2 到 f
%=	`g %= 9`	`g = g % 9`	3 到 g

圖 3.11　算術指定運算子

3.12 遞增和遞減運算子

- C還提供了單元遞增運算子 (increment operator)++ 和單元遞減運算子 (decrement operator)--。我們將這兩種運算子整理列成圖3.12。
- 如果變數c被遞增1的話，我們可用運算子++來代替運算式c=c+1或c+=1。
- 如果遞增或遞減運算子放在變數之前的話，他們稱為前置遞增 (preincrement) 或前置遞減運算子 (predecrement operators)。
- 如果遞增或遞減運算子放在變數之後的話，他們稱為後置遞增 (postincrement) 或後置遞減運算子 (postdecrement operators)。

3.12 遞增和遞減運算子

- 前置遞增（前置遞減）變數會使變數的值遞增（遞減）1，然後將變數新的值應用在該變數所出現的運算式當中。
- 後置遞增（後置遞減）一個變數會使變數目前的值用在該變數出現的運算式當中，然後該變數會遞增（遞減）1。

運算子	範例運算式	說明
++	++a	先將 a 遞增 1，再以 a 的新值進行運算
++	a++	以 a 目前的值進行運算，再將 a 遞增 1
--	--b	先將 b 遞減 1 再以 b 的新值進行運算
--	b--	以 b 目前的值進行運算，再將 b 遞減 1

圖 3.12　遞增和遞減運算子

3.12 遞增和遞減運算子

- 圖3.13的程式示範了前置遞增與後置遞增運算子的差異。
- 在此程式中，對變數 c 的後置遞增會使它在被 printf 敘述式使用之後，才加 1。
- 而對變數 c 的前置遞增，則會使它在被 printf 敘述式使用之前，便遞加 1。.

```c
/* Fig. 3.13: fig03_13.c
   Preincrementing and postincrementing */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int c; /* define variable */

   /* demonstrate postincrement */
   c = 5; /* assign 5 to c */
   printf( "%d\n", c ); /* print 5 */
   printf( "%d\n", c++ ); /* print 5 then postincrement */
   printf( "%d\n\n", c ); /* print 6 */

   /* demonstrate preincrement */
   c = 5; /* assign 5 to c */
   printf( "%d\n", c ); /* print 5 */
   printf( "%d\n", ++c ); /* preincrement then print 6 */
   printf( "%d\n", c ); /* print 6 */
   return 0; /* indicate program ended successfully */
} /* end function main */
```

圖 3.13　示範前置遞增和後置遞增的差異

```
5
5
6
  s
5
6
6
```

圖 3.13　示範前置遞增和後置遞增的差異

3.12 遞增和遞減運算子

- 此程式印出了變數c在使用++運算子之前及之後的值。
- 至於遞減運算子(--)的運作方式則與此相似。

良好的程式設計習慣 3.7

單元運算子應與其運算元緊密地寫在一起，中間不要留有空白。

3.12 遞增和遞減運算子

- 因此，圖3.10中的三個指定敘述式
  - ```
    passes = passes + 1;
    failures = failures + 1;
    student = student + 1;
    ```

  可以利用指定運算子改寫成
  - ```
    passes += 1;
    failures += 1;
    student += 1;
    ```

  也可以利用前置遞增運算子改寫成
  - ```
    ++passes;
    ++failures;
    ++student;
    ```

  或者利用後置遞增運算子改寫成
  - ```
    passes++;
    failures++;
    student++;
    ```

3.12 遞增和遞減運算子

- 有一點需注意的是，若是被遞增或遞減之變數位於一個只含有此變數的敘述式之內的話，那麼不論是前置遞增（減）或後置遞增（減），其效果是一樣的。

3.12 遞增和遞減運算子

- 只有當變數出現在長運算式中，前置遞增（減）和後置遞增（減）的效果才會不一樣。
- 到目前為止我們所學到的運算式中，只有單純的變數名稱才能做為遞增或遞減運算子的運算元。

常見的程式設計錯誤 3.10

將遞增或遞減運算子使用在一個運算式上，而不是一個單純的變數名稱。如 ++ (x+1) 便是個語法錯誤。

測試和除錯的小技巧 3.4

ANSI C 標準通常不會指定某一運算子之運算元的運算先後順序 (我們將在第 4 章看到有些例外狀況)。因此，你應該避免在同一敘述式中將某一變數遞增或遞減一次以上 (使用遞增或遞減運算子)。

3.12 遞增和遞減運算子

- 圖3.14列出了到目前為止，我們所介紹過之運算子的運算優先順序和結合性。
- 優先權順序是以表格的上方逐次往下遞減。
- 表中第二行描述了同一優先等級之運算子的結合性。
- 請注意，表中的條件運算子(?:)，一元的遞增(++)，遞減(--)，正(+)，負(-)及型別轉換運算子，以及指定運算子(=, +=, -=, *=, /=和%=)的結合性都是由右至左。
- 第三行則說明了這些運算子所屬的群組名稱。
- 圖3.14中其它所有運算子的結合性都是由左至右。

運算子	結合性	形式
++（後置） --（後置）	由右至左	後置
+ - (*type*) ++（前置） --（前置）	由右至左	單元性
* / %	由左至右	乘法
+ -	由左至右	加法
< <= > >=	由左至右	關係
== !=	由左至右	相等
?:	由右至左	條件
= += -= *= /= %=	由右至左	設值

圖 3.14　到目前為止所介紹之運算子的運算優先順序